DESIGNING AND IMPLEMENTING DATA STRUCTURE WITH SEARCH ALGORITHM TO SEARCH ANY ELEMENT FROM A GIVEN LIST HAVING CONSTANT TIME COMPLEXITY

Dr. Vimal P. Parmar [1] | Dr. CK Kumbharana [2]

[1] Research Scholar, Department of Computer Science, Saurashtra University Rajkot. Gujarat, India.
[2] Head Guide, Department of Computer Science, Saurashtra University Rajkot, Gujarat, India.

ABSTRACT

Search process is fundamental in computer science. To search an element various data structure are developed and implemented. Searching a word from a dictionary requires alphabetical arrangement of the words for fast searching. To search a record from a database an SQL query with WHERE clause is useful. To search specific information from internet, search engine is widely used. Searching is weaved centrally in almost all computer applications. This research paper is designed with available different search techniques and its time complexity. Various data structures are available with specific application needs. Linear search, binary search, binary search tree, hashing techniques and many other search techniques are studied and then prepared a hybrid data structure involving linked list with an array to directly locate an element. The process actually does not involve any searching technique to search element by comparison but using indexed organization of data structure makes it possible to determine whether an element exists in a list or not. The model is designed to prepare a data structure suitable for fast searching. Direct search can also be implemented using hashing techniques involving some mathematical computation to search any element in a constant time. Similarly, here is an effort is being made to search any element from a given list with constant time involving no mathematical computation to search a key like hashing technique. At last paper is concluded with limitations and future enhancement of the proposed technique.

KEYWORDS: Searching, Indexing, Linear Search, Binary Search, Binary Search Tree, Hashing Technique, Time Complexity, Space Complexity.

I. INTRODUCTION

Most of the computer applications involve searching. Searching an element from a list or a record from a database requires comparison. Hashing technique is a one that search an element directly by mapping a search key to an index position. Mapping is performed based on some mathematical calculation which can be implemented using any hash function. The main advantage of hashing is the time required to search any element is constant and is proportional to the time required by hashing function. Linear search technique requires scanning and comparing one by one element of a list with a search key. So the time required by linear search is proportional to the total number of elements in a list. If an element to be searched is located nearer to first element it requires less time as opposed to an element which is located at the end of the list or an element does not exist at all in the list requires maximum time because it requires maximum comparisons. The solution to this is the binary search which divides the search space half at each comparison giving fast performance compared to the linear search. Binary search tree is construction of a tree and traversing it left or right branch based on the condition. The performance is similar to that of binary search. Binary search requires elements to be arranged in some order to implement it. Here is an effort is being made to search an element in a constant time with no mathematical calculation by directly locating the search element if exists. To implement the search algorithm efficiently, a data structure is designed involving linked list, array and pointer. Entire searching process is implemented in C language and tested with sample data. At last paper is concluded with limitations and future enhancement to the proposed process. Time complexity is the time required by any algorithm where as space complexity is the amount of memory required to store data used by algorithm. Asymptotic notation Big O is used to analyze the algorithm by determining the key operations of the algorithm. Here proposed search process is compared and analyzed with existing searching algorithm using Big O notation.

II. LINEAR SEARCH ALGORITHM

Linear search algorithm searches an element by comparing the given key element to be searched with all the elements of the given list starting from first element. Following algorithm is a sequential or linear search algorithm.

ALGORITHM LINEAR_SEARCH (Vector V, Integer N, Integer X) : Given a vector V consisting of N elements and an element X to be searched from a vector V.

Table 1 : Linear Search Algorithm

```
1. [Search for the location of value X in vector V]
I = 1
Repeat WHILE I <= N
    IF V[I] = X
    THEN
          WRITE ('ELEMENT FOUND')
          RETURN (I)
     ELSE
          I = I + 1
2. WRITE ('ELEMENT NOT FOUND')
RETURN (0)
```

Above algorithm starts searching for a given value of X from first position of the vector. It compares the first element V[1] with X and if match is found it prints the message of found and returns the position where X is located in vector V otherwise increments value of I from 1 to 2 to compare next element. This process continues for N times and once match is found it returns the value of location I in vector V. If not match found after comparing all the elements in a vector V it prints the message of not found and returns index 0 indicating element X is not found in vector V.

The time required by linear search is proportional to total elements N in a vector V as key operation in this algorithm is comparison operation IF V[I] = X. So the time complexity for linear search is O(N). The best case is one where less comparisons is required that is element is located at first position and time required for this case is constant regardless of number of elements N which is denoted as O(1). The worst case is the case in which search element is located at Nth position which requires N comparison or an element does not exist in vector which also requires N comparisons. Generally algorithm analysis is performed by considering the worst case so here the time complexity for search process is O (N). Third case is the average case analysis as we don't know where the element is actually located so in average case using probability linear search requires O (N/2) time. But again it is proportional to N and asymptotic analysis of time ignores lower order terms and constants and considers only the higher order terms so it is proportional to N. So the linear search performs on time complexity of O (N). That means the time required by algorithm is based on the total elements in a vector. This is the simplest search process. To improve the performance other solution is required. Widely used search process is the binary search that dramatically reduces the time compared to with that of linear search.

III. BINARY SEARCH ALGORITHM

Binary search algorithm requires all the elements in a vector in ascending or descending order. It first searches the vector with its center element by comparing it with a given value. If match is found algorithm terminates by returning its position and if no match is found it reduces the search space to the half of the original size. If search element is less than the middle element then only first half of the vector is searched and in case if it is greater than the middle element then only second half of the vector is searched in next iteration. In either of the case the list to be searched is reduced to half at each iteration of the search process. Following lists the binary search algorithm.

ALGORITHM BINARY_SEARCH (Vector V, Integer N, Integer X) : Given a vector V consisting of N ordered elements arranged in ascending order and an element X to be searched from a vector V. The variables LOW, HIGH and MID denote the lower, higher and middle limits of search interval. This function returns the index of the vector element if the search is successful otherwise returns 0.

Table 2 : Binary Search Algorithm

```
1.[Initialize lower and higher limits]
LOW = 1
HIGH = N
2. [Perform search process by comparison]
Repeat WHILE LOW <= HIGH
        [Find middle element index of interval]
        MID = (LOW + HIGH) / 2
        [perform comparison]
        IF X < V[MID]
        THEN HIGH = MID – 1
        ELSE    IF X > V[MID]
                THEN LOW = MID + 1
                ELSE
                    WRITE ('ELEMENT FOUND')
                    RETURN (MID)
3. WRITE ('ELEMENT NOT FOUND')
RETURN (0)
```

Binary search has initial interval of lower limit index1and higher limit N. It computes middle element index using an expression MID = (LOW + HIGH) / 2. It than compares this middle element with search element X. To search an element from total N = 1024 elements it requires maximum 10 comparisons that is LOG (2^{10}). So time required by binary search algorithm is O (LOG (N)) here for example if N = 1024 = 2^{10} it requires 10 comparisons. If total number of elements to be searched becomes 2048 than also total comparison will be increased by just 1that is 11. So at each time number of elements to be searched doubles but the comparison increases bu1 only. So the binary search algorithm is more efficient than linear search algorithm.

IV. HASHING TECHNIQUE

Hash table search technique performs search operation independent of total number of elements in a list. In this technique position of a particular entry in a table is determined by the value of the key for that entry. Mapping is required to transform from a key value to a table location. This task is performed by hashing function. Hashing function should be efficient enough and may contain arithmetic, logical or mathematical operation to compute a location. This will avoid the comparison to search an element. Whenever searching is performed for a given key its location is computed using hash function where the element is stored if exists. So design of hash function is more important for fast searching. There may be chances of hash function which may compute the same location in a table for different key values which is known as collision. To solve this problem collision resolution techniques are proposed to overcome the problem of duplicate location for different key values. Two collision resolution techniques are open addressing and chaining. Following is the listing of different hash functions but it can be implemented as per the needs and it should be quite efficient so the performance criteria can be achieved.

A. Division method
B. Midsquare method
C. Folding method
D. Digit analysis
E. Length-dependent method
F. Algebraic coding

Time required by hashing function is independent of total number of entries in a table which was dependent on linear search and binary search. Hash technique requires time to locate any element from a table is a function of time required by a hash function.

V. OTHER SEARCH STRUCTURE

Variety of data structures are proposed to search an element efficiently. One such search structure is binary search tree in which all the lower key values are stored in a left branch sub tree and higher values are stored in right sub tree of a root element. This applies to each element of a tree. Search processed from root node and traverses left or right depending on the value of the key. So required time is equivalent of which is required by a binary search algorithm that is O (LOG (N)). Other variations of a search tree are height balanced tree, weight balanced tree and 2-3 tree which requires comparison and trie structure which stores and searches element based on the character positions in a given key value. Any of the technique can be quite useful depending on the application needs. In this paper a data structure with algorithm is proposed to search an element efficiently independent of total number of elements similar to hashing technique but by avoiding any mapping function.

Following sections defines data structure and algorithm to store and locate an element from the list.

VI. DATA STRUCTURE TO STORE AND SEARCH ELEMENT

Following data structure is created. It is similar to a linked list. Linked list contains two data fields and a link to a next node. At most two nodes are created one for positive value and second for negative value which is determined by a flag.

Table 3 : Data Structure to store data

```
RECORD STRUCTURE NODE
            INTEGER data;
            INTEGER flag;
            STRUCTURE NODE link
STUCTURE NODE *A[1000]  // POINTER ARRAY
```

Above linked list node structure contains data to store an integer value. If value stored in data is positive then flag is set to 1and if value stored in data is negative then flag is set to-1. At most two nodes exist for same index value. A is a pointer array of type node structure which created an index of data. For example if data value to be stored is X then it will be stored at A[X] location. Also if X is negative then its location is A[X] but its corresponding flag value is set to -1. This data structure avoids comparison to locate any data value X and directly it can be achieved through A[X].

VII. ALGORITHM TO CONSTRUCT THE INDEXED DATA STRUCTURE TO STORE DATA VALUE

Following algorithm creates a data structure to prepare index and store data values where data actually be stored at specific location.

ALGORITHM CREATE (N, A) : This algorithm creates an indexed table of total N elements. A is an array of pointer of type structure node defined in table 3. Pointer array A is initialized with NULL to all its entries. Also total number of entries in A can be allocated dynamically. FLG is a temporary variable to determine number is negative or positive. TMP is a temporary pointer variable to create second node for the same index entry. Negative index is not possible so if given value X < 0 then using X = -X it will be made positive and its flag value is set to -1. When an element is to be searched combined data * flag value is compared with search key to determine whether element exists in a list or not. This single comparison is required to locate any element. Where the element is located can directly be found using its index and in that index entry two numbers may be possible positive or negative and so it requires only one comparison whether the given element exists or does not exist. Following algorithm construct a table with N elements.

Table 4 : Algorithm To create indexed table

```
REPEAT FOR I = 1 TO N
    INPUT X
    FLG = 1
    IF X < 0
    THEN X = -X;
           FLG = 1
    IF A[X] = NULL
        A[X] = CREATE NEW NODE
        A[X] -> data = X
        A[X] -> flag = FLG
        A[X] -> link = NULL
   ELSE
        TMP = CREATE NEW NODE
        TMP -> data = X
        TMP -> flag = FLG
        TMP -> link = NULL;
        A[X] -> link = TMP;
```

VIII. ALGORITHM TO SEARCH AN ELEMENT FROM TABLE

Once the indexed table is constructed the search procedure becomes easier to implement. Following function searches the table using index X for any given value of X in a table.

Algorithm Function Search (A, X) : This function searches an element X in an indexed table A where A is an array of pointer of type structure node described in table 3. TMP is a temporary pointer variable of type structure node. Y is temporary integer variable to compare with indexed entry. On success it returns 1, 0 otherwise.

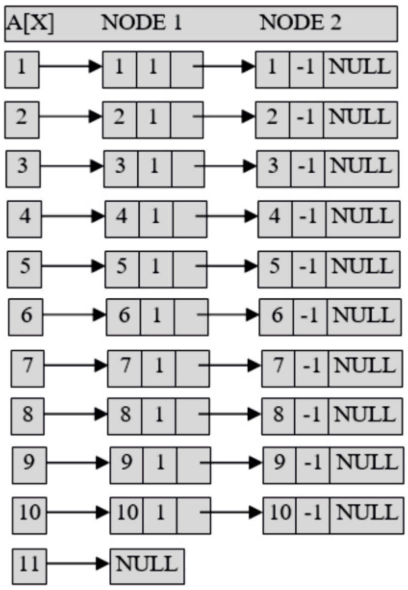
Table 5 : Algorithm To search indexed table

```
Y = X
IF X < 0
    X = -X
IF A[X] = NULL
    RETURN 0
ELSE
    TMP = A[X]
    REPEAT WHILE TMP != NULL
            IF (TMP->data)*(TMP->flag) = Y
            return 1
            TMP = TMP -> link
RETURN 0
```

First search value X is stored in temporary variable Y. If the given value X < 0 then it is made positive using X = -X. Then it will locate A[X] directly. If it is NULL using condition A[X] = NULL means that location is empty and hence element does not exist in a list so it returns 0. Otherwise it contains address pointing to structure node which is stored in pointer variable TMP. At most two nodes exists one for negative and one for positive value of X which is determined by flag value. It traverses linked list and compares the given key value Y with node value using an expression (TMP->data)*(TMP->flag) and if match is found it returns 1. Loop terminates when TMP becomes NULL and at last it returns 0. So maximum comparison it takes is two for an element to search in a list.

IX. SAMPLE DATA SET FOR PROPOSED ALGORITHM

The entire proposed process is implemented and tested. Following example demonstrates how the indexed table is propagated for 1to 10 negative and positive values



Array of pointer starting from 1to as many numbers of table size can be created dynamically and it is initialized with NULL. So at beginning array A has no elements and all elements A[1], A[2] and so on contains NULL. Once an entry with value 1 is to be entered, anew node is created with data =1 and flag = 1with link = NULL and its address is assigned to A[1]. Now value -1is to be stored then its index is 1, so again new node is created with data =1 and flag = -1 with link = NULL and its address is stored at link field of node with value 1. So whenever value X is to be searched it is directly be searched at location A[X] and traversing linked list having at most two elements X and –X. By comparing the given value with node value using expression data * flag determines whether the given element resides in a list or not. All the entries with no data will be NULL.

Time complexity for this process is constant with maximum of two comparisons which is necessary because negative and positive numbers are stored at same location index. Space complexity is high here. Time and space are reverse proportional to each other. If we want fast performance by reducing time then inadvertently memory requirement increases. Here for this procedure space complexity is greater to achieve the desired constant time performance. Today due to latest technology and hardware advancement memory availability is easier but it can further be optimized to reduce the memory needs. This direct searching technique searches an element from a list with any number of total elements in a constant time and time required to search any element is almost same for every element.

X. CONCLUSION

To achieve the desired time complexity here this process requires more space complexity. This process is demonstrated only for the integer values to be stored and searched but it can be expanded and improved to implement for any kind of data values. Text and string may be required to process to uniquely identify its index entry. To store and retrieve any integer with negative and positive values this searching technique performs in a constant time. Further memory space can be optimized to reduce the memory needs.

REFERENCES

[1] An Introduction to data structures with applications - (Mcgraw Hill Computer Science Series) [Jean-Paul Tremblay, Paul G. Sorenson, P. G. Sorenson]

[2] Comparing Linear Search and Binary Search Algorithms To Search An Element From A Linear List Implemented Through Static Array, Dynamic Array And Linked List - International Journal of Computer Applications (IJCA) 121(3) 13-17 -July 2015 Volume 121/ Number 3 DOI : 10.5120/21519-4495

[3] Programming with ANSI C – E. Balagurusamy

[4] Let Us C – Yashwant Kanitkar

[5] C Programming 2nd Edition – Dennis M. Ritchie and Brian W. Kernighan